T.P. 05

Parcours d'échiquier

■ ■ ■

Les sujets et leur corrigés sont maintenant accessibles à l'adresse : http://www.pps.univ-paris-diderot.fr/~pboutill/prepa/.

Partie I : Infrastructure

I - Mouvements du cavalier

On s'intéresse aux mouvements d'un cavalier dans un plan à deux dimensions. Les mouvements du cavalier sont représentés par des vecteurs $(int * int)$, et seuls les mouvements valides aux échecs (en "L") sont autorisés.
Le sujet n'est pas récent puisqu'en 1776 Euler s'y interressait déja ! De plus amples informations et liens à ce propos sont disponible sur : http://images.math.cnrs.fr/Caracolades-sur-echiquiers.html.

Exercice 1. Créer une liste mouvements : $(int \times int)\ list$ des mouvements possibles d'un cavalier.

Exercice 2. Écrire ensuite la fonction deplace : $(int \times int) \to (int \times int) \to (int \times int)$ qui, étant donné une position de départ et un mouvement, renvoie les coordonnées de la position d'arrivée.

II - Gestion de l'échiquier

On représente l'échiquier par une matrice de booléens *bool vect vect* : echiq.(x).(y) vaudra true si et seulement si la case (x,y) est libre, c'est à dire que le cavalier n'y est pas encore passé. Ce ne sera pas forcément un échiquier standard (dimensions 8x8), ni forcément un échiquier carré. On essaiera d'avoir un algorithme le plus performant possible, c'est à dire pouvant nous apporter une réponse sur les plus gros échiquiers possibles (40x40 par exemple).

Exercice 3. Écrire une fonction valide : $(int \times int) \to (int \times int) \to bool$ qui, étant donné les dimensions (n,p) d'un échiquier, indique si le couple (x,y) donné en deuxième paramètre est bien un couple d'indices valide (qui ne sort pas de la matrice)

Exercice 4. Écrire une fonction filter : $('a \to bool) \to 'a\ list \to 'a\ list$ qui, étant donné une liste li : $'a\ list$ et un prédicat p : $'a \to bool$, renvoie la liste des éléments x de li tels que p x = true.

Exercice 5. En déduire une fonction deplacements_valides : $(int \times int) \to (int \times int) \to (int \times int)\ list$ qui, étant donné les dimensions de l'échiquier et un point de départ du cavalier, renvoie la liste des positions valides qu'il peut atteindre en un seul mouvement.
Indice : vous pourrez utiliser la fonction map.
La case (x,y) de l'échiquier echiq est echiq.(x).(y). Pour les algorithmes qui vont suivre, vous êtes encouragés à utiliser des fonctions auxiliaires pour pouvoir lire et modifier les cases de l'échiquier sans avoir à décomposer le couple position à chaque fois :

value get : $'a\ vect\ vect \to (int \times int) \to' a$

value set : $'a\ vect\ vect \to (int \times int) \to' a \to unit$

Partie II : Recherche

III - Existence d'une solution

Dans un premier temps, on essaiera seulement de savoir si un solution (un chemin qui passe une et une seule fois par chaque case de l'échiquier) existe, sans essayer de donner le chemin correspondant : c'est plus simple à coder.

On utilise pour cela la méthode du *backtracking* (retour en arrière) : pour trouver le bon chemin, on essaie tous les mouvements possibles les uns après les autres et, si on se retrouve bloqué, on revient en arrière en annulant certain mouvements pour explorer d'autres possibilités.
On met en place un algorithme récursif : étant donné une position libre de l'échiquier (position courante), le nombre de cases déjà parcourues, et des informations sur les cases déjà parcourues (une case est à false si elle a déjà été parcourue, true si on peut encore y aller),

* si toutes les cases sauf une (la position courante) ont déjà été parcourues, on a gagné et on renvoie true
* sinon
  - on marque la position courante comme occupée
  - on regarde si l'un des déplacements possibles depuis la position courante vers une case libre est une solution (appels récursifs)
  - si aucun de ces essais n'a trouvé de solution, on revient en arrière : on efface le chemin emprunté (en marquant la position courante comme libre) et on renvoie false

Exercice 6. Trouver dans la documentation Caml Light une fonction qui permet de savoir si une liste contient au moins un élément satisfaisant la condition que l'on souhaite (ici, on voudra savoir si l'appel de notre algorithme sur une des destinations possibles du cavalier renvoie true, mais on veut juste une fonction générique pour l'instant).
Si vous ne l'avez pas trouvée, codez-la vous-mêmes. Mais c'est mal !

```
let resoudre dimensions =
  let (n, p) = dimensions in
  let echiquier = make_matrix n p true in
  let rec parcours nb_parcourues pos =
    (* votre code ici *)
  in parcours 0 (0, 0);;
```

Exercice 7. Comprendre, puis implémenter cet algorithme. On pourra utiliser le squelette de code ci-contre.
Tester votre algorithme sur de petits échiquiers : 3x3, 4x4, 4x5, 6x6, 8x8.

IV - Trouver le chemin solution

C'est très amusant de savoir que "oui, il est possible de parcourir un échiquier 8x8 avec un cavalier sans repasser deux fois par la même case", mais on aimerait bien que notre algorithme nous donne en plus la liste des déplacements du cavalier.
On utilisera pour cela une deuxième matrice, successeur : $(int \times int)\ vect\ vect$, qui stockera pour chaque case du chemin la case suivante sur laquelle s'est rendu le cavalier. Quand on a trouvé une solution, il suffit alors de suivre les informations de la matrice successeur en partant de la case de départ pour reconstituer le chemin complet du cavalier sur l'échiquier.

Exercice 8. Modifier votre algorithme de parcours pour qu'à chaque fois qu'on essaie une case depuis la position courante, la valeur associée à la position courante dans successeur soit mise à jour.

Exercice 9. Écrire une fonction chemin : $(int \times int)\ vect\ vect \rightarrow (int \times int) \rightarrow (int \times int)\ list$ qui, étant donné un tableau successeur et une position de départ, renvoie la liste des cases parcourues successivement.
L'intégrer à votre fonction resoudre pour renvoyer le chemin solution quand il existe (et sinon par exemple la liste vide).

Exercice 10.

Question facultative : écrire une fonction affiche_chemin qui affiche le chemin solution d'une manière agréable ; il faudrait que l'on puisse vérifier "à l'oeil" que c'est un chemin valide et qui remplit l'échiquier.
On pourra prendre en paramètre les dimensions de la matrice, ou toute autre information facilement accessible à l'utilisateur, si on le souhaite.

Partie III : Heuristiques

Comme vous le constatez sans doute, votre code marche (ou pas) mais il est très lent. C'est normal, puisqu'on explore tous les chemins possibles et que leur nombre croit exponentiellement avec les dimensions de l'échiquier.
En remarquant que l'algorithme s'arrête dès qu'il trouve une solution, au lieu de continuer à chercher toutes les solutions (vérifiez-le), on va essayer de le modifier un peu pour que la solution soit parmis les premiers chemins explorés. Autrement dit, on voudrait qu'au moment de choisir les voisins de la position courante (les cases libres à un mouvement de distance) à explorer, on essaie en premier les cases les "meilleures" (celles qui ont le plus de chance de faire partie du chemin solution).
Le choix que l'on va faire ici est de choisir en premier les cases ayant le moins de voisins libres. En quelque sorte, il vaut mieux y passer le plus vite possible, parce qu'elles sont plus difficiles à atteindre.

Exercice 11. Modifier l'algorithme pour qu'il essaie en premier les cases voisines qui ont elles-mêmes le moins de voisins libres.
Vous aurez sans doute besoin d'une fonction pour trier une liste selon le critère votre choix. Cherchez-la dans la documentation (standard libraries).

Exercice 12. Améliorer un peu le critère de sélection : si deux cases sont à égalité (elles ont aussi peu de voisins libres l'une que l'autre), on choisit celle qui est la plus proche du bord de l'échiquier.

Partie IV : Question difficile

Exercice 13. Écrire un *quine*, un programme qui affiche exactement son propre code en sortie.